

From Byzantine Failures to Crash Failures in Message-Passing Systems: a BG Simulation-based approach

Damien Imbs[†], Michel Raynal^{*,‡}, Julien Stainer[•]

^{*} Institut Universitaire de France

[†] Department of Mathematics, University of Bremen, Germany

[‡] IRISA, Université de Rennes, 35042 Rennes, France

[•] École Polytechnique Fédérale de Lausanne, Switzerland

imbs@uni-bremen.de raynal@irisa.fr julien.stainer@epfl.ch

Abstract

The BG-simulation is a powerful reduction algorithm designed for asynchronous read/write crash-prone systems. It allows a set of $(t + 1)$ asynchronous sequential processes to wait-free simulate (i.e., despite the crash of up to t of them) an arbitrary number n of processes under the assumption that at most t of them may crash. The BG simulation shows that, in read/write systems, the crucial parameter is not the number n of processes, but the upper bound t on the number of process crashes.

The paper extends the concept of BG simulation to asynchronous message-passing systems prone to Byzantine failures. Byzantine failures are the most general type of failure: a faulty process can exhibit any arbitrary behavior. Because of this, they are also the most difficult to analyze and to handle algorithmically. The main contribution of the paper is a signature-free reduction of Byzantine failures to crash failures. Assuming $t < \min(n', n/3)$, the paper presents an algorithm that simulates a system of n' processes where up to t may crash, on top of a basic system of n processes where up to t may be Byzantine. While topological techniques have been used to relate the computability of Byzantine failure-prone systems to that of crash failure-prone ones, this simulation is the first, to our knowledge, that establishes this relation directly, in an algorithmic way.

In addition to extending the basic BG simulation to message-passing systems and failures more severe than process crashes, being modular and direct, this simulation provides us with a deeper insight in the nature and understanding of crash and Byzantine failures in the context of asynchronous message-passing systems. Moreover, it also allows crash-tolerant algorithms, designed for asynchronous read/write systems, to be executed on top of asynchronous message-passing systems prone to Byzantine failures.

Keywords: Asynchronous processes, BG simulation, Byzantine process, Distributed computability, Fault-tolerance, Message-passing system, Process crash, Read/write shared memory system, Reduction algorithm, t -Resilience, System model, Wait-freedom.

1 Introduction

What is the Borowsky-Gafni (BG) simulation and why is it important? Considering an asynchronous system where processes can crash, the (n, k) -set agreement problem is a basic distributed decision task defined as follows [11]. Each of the n processes proposes a value, and every process that does not crash has to decide a value (termination), such that a decided value is a proposed value (validity) and at most k different values are decided (agreement). The consensus problem corresponds to the particular case $k = 1$.

The (n, k) -set agreement is fundamental because it captures the essence of fault-tolerant distributed computability issues. A central question related to asynchronous distributed computability is the following: “Can we use a solution to the (n, k) -set agreement problem as a subroutine to solve the (n', k') -set agreement problem, when at most $t < \min(n, n')$ processes may crash?” (“Is (n', k') -set agreement reducible to (n, k) -set agreement?”). The BG simulation (initially sketched in [6] and then formalized in a journal version [7], where, in addition, a formal definition of “reducibility” is given) answers this fundamental question. It states that the answer is “yes” if $k' \geq k$ and “no” if $k' \leq t < k$. As we can see, the answer “yes” does not depend on the number of processes.

To this end, the algorithm described in [7] allows $(t + 1)$ processes to simulate a large number n' of asynchronous processes that communicate through read/write registers, and collectively solve a decision task, in the presence of at most t crashes. Each of the $(t + 1)$ simulator processes simulates all the n' processes. These $(t + 1)$ simulator processes cooperate through underlying objects that allow them to agree on a single output for each of the non-deterministic statements issued by every simulated process. (These underlying objects, called safe agreement objects, can be built of top of read/write atomic registers.)

Let $\text{BG}(\text{RW}, \text{C})$ denote the basic BG simulation algorithm [7] (RW stands for “read/write communication”, and C stands for “crash failures”). $\text{BG}(\text{RW}, \text{C})$ is “symmetric” in the sense that each of the n' processes is simulated by every simulator, and the $(t + 1)$ simulators are “equal” with respect to each simulated process, namely, (1) every simulator fairly simulates all the processes, and (2) the crash of a simulator entails the crash of at most one simulated process. This symmetry allows $\text{BG}(\text{RW}, \text{C})$ to be suited to colorless tasks (i.e., distributed computing problems where the value decided by a process can be decided by any process [17]). $\text{BG}(\text{RW}, \text{C})$ has then been extended to colored tasks (i.e., tasks such as renaming [3], where a process cannot systematically borrow its output from another process). Extended BG simulation is addressed in [14, 22]. Algorithmic pedagogical presentations of the BG simulation can be found in [18, 22]. A topological view on distributed computability issues in Byzantine asynchronous message-passing systems has been recently presented in [16, 28]. A pedagogical topology-based presentation of the BG-simulation is given in chapter 7 of [16].

What is learned from the BG simulation The important lesson learned from the BG simulation is that, in a failure-prone context, what is important is not the number of processes but the maximal number of possible failures and the actual number of values that are proposed to a decision task. An interesting consequence of the BG simulation (among several of its applications [7]) is the proof that there is no t -resilient (n, k) -set agreement algorithm for $t \geq k$. This is obtained as follows. As (1) the BG simulation allows reducing the $(k + 1, k)$ -set agreement problem to the (n, k) -set agreement problem in a system with up to k failures, and (2) the $(k + 1, k)$ -set agreement problem is known to be impossible in presence of k failures [6, 19, 30], it follows that there is no k -resilient (n, k) -set agreement algorithm.

Content of the paper: on the BG-simulation side As already indicated, the BG simulation has been explored in asynchronous systems where processes (1) communicate through atomic read/write registers [25], and (2) may commit only crash failures. This paper extends it in two directions. The first is the communication model, namely, it considers that processes cooperate by sending and receiving messages via asynchronous reliable channels. The second dimension is related to the type of failures; more precisely, it considers two

types of failures: process crash failures, and the more severe process Byzantine failures. The paper presents the following contributions.

A first is an algorithm, denoted BG(MP,C), which simulates the execution of a colorless task running in an asynchronous message-passing system of n' processes, where up to t may crash, on top of an asynchronous message-passing system of n processes where up to t may crash. This simulation requires $t < n/2$ (which is a necessary and sufficient condition to simulate read/write registers in asynchronous message-passing systems of n processes [2]). While the number of simulated processes n' can be any integer, for the simulation to be non-trivial we consider that $t < n'$.

A second contribution is an algorithm, denoted BG(MP,B), which simulates the execution of a colorless task running in an asynchronous message-passing system of n' processes, where up to t may crash, on top of an asynchronous message-passing system of n processes where up to t may be Byzantine [26]. This simulation requires $t < n/3$ (according to the task which is simulated, additional constraint on t may be needed, see [16]; see also Section 6). As in the case of BG(MP,C), and for the same reason, we consider that $t < n'$. This algorithm has two noteworthy features: it is the first BG simulation algorithm that considers Byzantine failures, and it allows to run a crash-tolerant algorithm solving a colorless task on top of an asynchronous system prone to Byzantine failures. Both the algorithms BG(MP,C) and BG(MP,B) are *genuine* in the sense they do not rely on the simulation of an underlying shared memory.

While the full-information algorithm presented in [28] can be used to decide when there is a simulation between two models, the present paper is the first (to our knowledge) that allows the direct execution in the presence of Byzantine failures of any crash-tolerant algorithm that solves a colorless task. BG(MP,B) provides an algorithmic approach which complements the topology-based simulation framework of [28], and may also be of practical interest. It has the interesting property that the simulation of a message only requires a polynomial number of messages in the base system, and the increase in size of these messages, when compared to the size of the simulated message, is also polynomial. Additionally, differently from early works on Byzantine failures like [15], it does not use any cryptography-based mechanism.

Content of the paper: on the safe agreement objects side The core of the previous algorithms lies in new underlying safe agreement objects, which allow the n simulators to agree on the next operation executed by each of the n' simulated processes. Such a safe agreement object ensures that all the simulators produce the very same simulation. At the operational level, a safe agreement object provides processes with two operations, denoted `propose()` and `decide()`, which are invoked in this order by each correct process. The termination property associated with a safe agreement object SA is the following: if no simulator commits a failure while executing $SA.propose()$, then any invocation of $SA.decide()$ by a non-faulty simulator terminates. Moreover, no two correct processes decide differently.

On the algorithmic side, a novelty of the paper lies in the algorithms implementing these new safe agreement objects. Differently from their read/write memory counterparts, they are not based on underlying snapshot objects [1]. They instead rely heavily on message communication patterns inspired from the reliable broadcast algorithms introduced in [8].

A last and noteworthy contribution of the paper lies in the second algorithm (which implements safe agreement in a Byzantine message-passing system). This object is the core of a simulation when one wants to execute asynchronous read/write crash-tolerant algorithms on top of asynchronous message-passing systems prone to Byzantine failures.

Existing simulations considering Byzantine failures Simulations of crash failures in a Byzantine system have been addressed in the context of synchronous systems [5, 29, 31]. The only articles we are aware of concerning such a simulation in asynchronous systems are [12, 16, 20]. As noticed in [4], [12] considers a restricted class of round-based deterministic algorithms. The simulation presented in [16] executes a full-information asynchronous crash-tolerant algorithm in an asynchronous Byzantine failure-prone system. The

article [20] considers an agent/host model and focuses mainly on reliable broadcast.

Roadmap The paper is composed of 6 sections. Section 2 presents both the crash-prone and the Byzantine asynchronous message-passing models, and the notion of a task. Section 3 presents the structure of the simulation algorithms. Section 4 presents the simulation algorithm BG(MP,C), while Section 5 presents the simulation algorithm BG(MP,B). Finally, Section 6 addresses the computability implications of the Byzantine-tolerant simulation and its underlying safe agreement object.

2 Computation Models and Tasks

2.1 Computation models

Computing entities The system is made up of a set Π of n sequential processes, denoted p_1, p_2, \dots, p_n . These processes are asynchronous in the sense that each process progresses at its own speed, which can be arbitrary and remains always unknown to the other processes.

During an execution, processes may deviate from their specification. In that case, the corresponding processes are said to be *faulty*. A process that does not deviate from its specification is *correct* (or *non-faulty*). The model parameter t denotes the maximal number of processes that can be faulty in a given execution. Two failure types are considered below.

Communication model The processes cooperate by sending and receiving messages through bi-directional channels. The communication network is a complete network, which means that each process p_i can directly send a message to any process p_j (including itself). Each channel is reliable (no loss, corruption, or creation of messages), not necessarily first-in/first-out, and asynchronous (while the transit time of each message is finite, there is no upper bound on message transit times).

The macro-operation “broadcast TYPE(m)”, where TYPE is a message type and m is its content, is a shortcut for the following statement: “send TYPE(m) to each process (including itself)”.

The process crash failure model In the crash failure model, a process may prematurely stop its execution. A process executes correctly its algorithm until it possibly crashes. Once crashed, a process remains crashed forever. It is assumed that at most t processes may crash. If there is no specific constraint on t , the corresponding model is denoted $\mathcal{CAMP}_{n,t}[t < n]$. When it is assumed that at most $t < n/2$ processes may crash, the corresponding model is denoted $\mathcal{CAMP}_{n,t}[t < n/2]$.

The Byzantine failure model A Byzantine process is a process that behaves arbitrarily: it may crash, fail to send or receive messages, send arbitrary messages, start in an arbitrary state, perform arbitrary state transitions, etc. Hence, a Byzantine process, which is assumed to send the same message m to all the processes, can send a message m_1 to some processes, a different message m_2 to another subset of processes, and no message at all to the other processes. Moreover, Byzantine processes can collude to “pollute” the computation.

It is assumed that Byzantine processes cannot control the network, hence, when a process receives a message, it can unambiguously identify its sender. As previously, t denotes the upper bound on the number of processes that may commit Byzantine failures. If there is no constraint on t , the corresponding model is denoted $\mathcal{BAMP}_{n,t}[t < n]$. When it is assumed that at most $t < n/3$ processes may be faulty, the corresponding model is denoted $\mathcal{BAMP}_{n,t}[t < n/3]$.

2.2 Decision tasks and algorithms solving a task

Decision tasks The problems we are interested in are called *decision tasks* (the reader interested in a more formal presentation of decision tasks can consult the literature, e.g., [7, 19]). In every run, each process proposes a value and the proposed values define an input vector I , where $I[j]$ is the value proposed by process p_j . Let \mathcal{I} denote the set of allowed input vectors. Each process has to decide a value. The decided values define an output vector O , such that $O[j]$ is the value decided by p_j . Let \mathcal{O} be the set of the output vectors.

A decision task is a binary relation Δ from \mathcal{I} into \mathcal{O} . A task is *colorless* if, when a value v is proposed by a process p_j (i.e., $I[j] = v$), then v can be proposed by any number of processes and, when a value v' is decided by a process p_j (i.e., $O[j] = v'$), then v' can be decided by any number of processes. Consensus, and more generally k -set agreement, are colorless tasks. Otherwise the task is *colored*. Symmetry breaking and renaming are colored tasks [3, 10, 21].

Algorithm solving a task An algorithm solves a task in a t -resilient environment if, given any $I \in \mathcal{I}$, (1) each correct process p_j decides a value o_j , and (2) there is an output vector O such that $(I, O) \in \Delta$ where O is defined as follows. If p_j decides o_j , then $O[j] = o_j$. If p_j does not decide, $O[j]$ is set to any value v' that preserves the relation $(I, O) \in \Delta$.

Considering a system of n processes, a task is t -resiliently solvable if there is an algorithm that solves it in the presence of at most t faulty processes. As an example, consensus is not 1-resiliently solvable in asynchronous crash-prone systems, be the communication medium a set of read/write registers [27], or a message-passing system [13]. Differently, renaming with $2n - 1$ new names is $(n - 1)$ -resiliently solvable in asynchronous read/write crash-prone systems [9, 19], and is t -resiliently solvable in asynchronous crash-prone message-passing systems for $t < n/2$ [3].

3 Structure of the Simulation Algorithms

Aim Let A' be an algorithm that solves a colorless decision task among n' processes in the system model $\mathcal{CAMP}_{n',t}[t < n']$. The aim is to design an algorithm that simulates A' in the system model $\mathcal{CAMP}_{n,t}[t < n/2]$ (resp., $\mathcal{BAMP}_{n,t}[t < n/3]$). As already indicated, the corresponding simulation algorithm is denoted BG(MP,C) in the first case, and BG(MP,B) in the second case.

Notation A simulated process is denoted p_j , where $1 \leq j \leq n'$. Similarly, a simulator process (“simulator” in short) is denoted q_i , where $1 \leq i \leq n$. The set Π denote the set of the simulator indexes, i.e., $\Pi = \{1, \dots, n\}$.

The safe agreement objects, build in the simulation and used by the simulators, are identified with upper case letters, e.g., SA . The variables local to simulator q_j is identified with lower case letters, and the resulting identifiers are subscripted with j .

Behavior of a simulator q_i Each simulator is given the code of all the simulated processes $p_1, \dots, p_{n'}$. It manages n' threads, one associated with each simulated process, and executes them in a fair way.

The code of a simulated process p_j contains local statements, send statements, and receive statements. It is assumed that the behavior of a simulated process p_j is deterministic in the sense it is entirely defined from its local input (as defined by the task instance), and the order in which p_j receives messages.

The simulation has to ensure that (1) all simulators simulate the same behavior of the set of simulated processes, and (2) a faulty simulator entails the failure of at most one simulated process. The way this is realized depends, of course, on the failure model that is considered.

4 BG(MP,C): BG in the Crash-prone Asynchronous Message-Passing Model

This section presents the algorithm BG(MP,C). As previously indicated, this algorithm simulates, in the model $\mathcal{CAMP}_{n,t}[t < n/2]$, an algorithm A' solving a task in $\mathcal{CAMP}_{n',t}[t < n']$. It is made up of two parts: an algorithm implementing a safe agreement object, and the simulation itself, which uses several of these objects to allow the simulators to cooperate.

4.1 Safe agreement object in $\mathcal{CAMP}_{n,t}[t < n/2]$: definition

This object type (or variants of it), briefly sketched in the Introduction, is at the core of both the BG simulation [6, 7, 14, 22], and the liveness guarantees of concurrent objects [23, 24]. It is a one-shot object that solves consensus in failure-free scenarios, and allows processes to agree with a weak termination guarantee in the presence of failures.

A safe agreement object provides each simulator q_i , $1 \leq i \leq n$, with two operations denoted `propose()` and `decide()`, that q_i can invoke at most once, and in this order; `propose()` allows q_i to propose a value, while `decide()` allows it to decide a value. Considering the crash failure model, the properties associated with this object are the following ones.

- **Validity.** A decided value is a proposed value.
- **Agreement.** No two simulators decide distinct values.
- **Propose-Termination.** An invocation of `propose()` by a correct simulator terminates.
- **Decide-Termination.** If no simulator crashes while executing `propose()`, then any invocation of `decide()` by a correct simulator terminates.

It is easy to see that a safe agreement object is a consensus object whose termination condition is failure-dependent. Algorithms implementing safe agreement objects (or variants of it) can be found in [6, 7, 24].

4.2 Safe agreement object in $\mathcal{CAMP}_{n,t}[t < n/2]$: algorithm

An algorithm implementing a safe agreement object in $\mathcal{CAMP}_{n,t}[t < n/2]$ is described in Figure 1.

Local data structures Each simulator q_i , $1 \leq i \leq n$, manages three local data structures, namely, the arrays $values_i[1..n]$, $my_view_i[1..n]$, $all_views_i[1..n]$, all initialized to $[\perp, \dots, \perp]$, where \perp denotes a default value that cannot be proposed to the safe agreement object by the simulators.

- The aim of $values_i[x]$ is to contain, as currently known by q_i , the value proposed to the safe agreement object by the simulator q_x .
- The aim of $my_view_i[x]$ is to contain, as known by q_i , the value proposed to the safe agreement object by the simulator q_x , as witnessed by strictly more than $\frac{n}{2}$ distinct simulators (i.e., at least a correct process).
- The aim of $all_views_i[x]$ is to contain what q_i 's knows about the view seen by q_x .

Algorithm: the operation `propose()` The algorithm implementing the operation `propose()` invoked by a simulator q_i is described at lines C01-C14 (client side) and lines C20-C22 (server side). This algorithm is made up of three parts.

First part. A simulator q_i first broadcasts the message `VALUE` (i, v_i) , where v_i is the value it proposes to the safe agreement object (line C01). Then, it waits until it knows that strictly more than $\frac{n}{2}$ simulators know its value (line C02). On its “server” side, when q_i receives for the first time the message `VALUE` (x, v) , it

first saves v in $values_i[x]$; then it forwards the received message to cope with the (possible) crash of q_x (this witnesses the fact that q_i knows the value proposed by p_x , line C20)¹).

```

operation propose ( $v_i$ ) is
(C01) broadcast VALUE ( $i, v_i$ );
(C02) wait (VALUE ( $i, v_i$ ) received from strictly more than  $\frac{n}{2}$  different simulators);
(C03) for each  $x \in [1..n]$  do broadcast READ ( $i, x$ ) end for;
(C04) for each  $x \in [1..n]$  do
(C05)   wait (READ'ANSWER ( $i, x, \perp$ ) received from strictly more than  $\frac{n}{2}$  different simulators
(C06)      $\vee \exists w : \text{VALUE} (x, w)$  received from strictly more than  $\frac{n}{2}$  different simulators);
(C07)   if (predicate of line C06 satisfied)
(C08)     then  $my\_view_i[x] \leftarrow w$ 
(C09)     else  $my\_view_i[x] \leftarrow \perp$ 
(C10)   end if
(C11) end for;
(C12) broadcast VIEW ( $i, my\_view_i$ );
(C13) wait (VIEW ( $i, my\_view_i$ ) received from strictly more than  $\frac{n}{2}$  different simulators);
(C14) return().

operation decide () is
(C15) wait ( $\exists$  a non-empty set  $\sigma \subseteq \Pi$ :
(C16)    $\forall y \in \sigma : [(all\_views_i[y] \neq \perp) \wedge (\forall z \in \Pi : (all\_views_i[y][z] \neq \perp) \Rightarrow (z \in \sigma))]$ ;
(C17) let  $min\_sigma_i$  be the set  $\sigma$  of smallest size;
(C18) let  $res$  be  $\min(\{values_i[y] : y \in min\_sigma_i\})$ ;
(C19) return( $res$ ).

%-----
when the message VALUE ( $x, v$ ) is received for the first time:
  % "for the first time" is with respect to each pair of values ( $x, v$ ) %
(C20)  $values_i[x] \leftarrow v$ ; broadcast VALUE ( $x, v$ ).

when the message READ ( $j, x$ ) is received for the first time:
(C21) send READ'ANSWER ( $j, x, values_i[x]$ ) to  $q_j$ .

when the message VIEW ( $x, view$ ) is received for the first time:
(C22)  $all\_views_i[x] \leftarrow view$ ; broadcast VIEW ( $x, view$ ).

```

Figure 1: Safe agreement object in $\mathcal{CAM}\mathcal{P}_{n,t}[t < n/2]$ (code for the simulator q_i)

Second part. In this part, q_i builds a local view of the values proposed by the n simulators. To this end, it first broadcasts messages READ (i, x), $1 \leq x \leq n$, to learn the value proposed by each simulator q_x (line C03). On its server side, when q_i receives such a message, it broadcasts by return its current knowledge of the value proposed by q_x (line C21).

Then, the simulator q_i builds its local view of the values that have been proposed. For each simulator q_x , q_i waits until it has received from strictly more than $\frac{n}{2}$ distinct simulators the very same message, namely, either the message READ'ANSWER (i, x, \perp), or the message VALUE (x, w) (lines C05-C06). In the first case, q_i considers that q_x has not yet proposed a value, while in the second case it considers that q_x proposed the value w (let us observe that, while q_i can receive both READ'ANSWER (i, x, \perp) and messages VALUE (x, w), it stops waiting as soon as it received strictly more than $\frac{n}{2}$ of one of them) (lines C07-C10).

Third part. Finally, the simulator q_i informs the other simulators on its local view $my_view_i[1..n]$. To this end, it broadcasts the message VIEW (i, my_view_i). When it has received the corresponding “acknowledgments”, q_i returns from its invocation of the operation propose() (line C12-C14). (The behavior of q_i when it

¹ Let us observe that the lines C01 and C20 implement a reliable broadcast of the message VALUE (i, v_i). Similarly, the lines C12 and C22 implement a reliable broadcast of the message VIEW (i, my_view_i). It is easy to see that the cost of such a reliable broadcast is $O(n^2)$ messages.

receives a message VIEW $(x, view)$ is similar to the one when it receives a message VALUE (x, v) . The only difference is that $values_i[x]$ is now replaced by $all_views_i[x]$, line C22.)

Algorithm: the operation decide() The algorithm implementing the operation decide() is described at lines C15-C19. It consists in a “closure” computation. A simulator q_i waits until it knows a non-empty set of simulators σ such that (a) it knows their views, and (b) this set is closed under the relation “has in its published view the value of” which means that the processes whose values appear in a view of a process of σ are also in σ (lines C15-C16).

Let us observe that it is possible that, locally, several sets satisfy this property. If it is the case, q_i selects the smallest of them. Let min_sigma_i be this set of simulators (lines C17). The value that is returned by q_i is then the smallest value among the values proposed by the simulators in min_sigma_i (lines C18-C19).

4.3 Safe agreement object in $\mathcal{CAMP}_{n,t}[t < n/2]$: proof

This section proves that the algorithm presented in Figure 1 implements a safe agreement object, i.e., any of its runs in $\mathcal{CAMP}_{n,t}[t < n/2]$ satisfies the validity, agreement, and termination properties, which define it.

Lemma 1. *An invocation of propose() by a simulator that does not crash during this invocation, terminates.*

Proof Let us consider a simulator q_i that does not crash during its invocation of propose(). Hence, q_i broadcast the message VALUE (i, v_i) at line C01. This message is received by strictly more than $\frac{n}{2}$ correct simulators, and each of them broadcasts this message when it receives it. It follows that q_i cannot block forever at line C02.

Let us now consider the wait statement at lines C05-C06. There are two cases. Let READ (i, x) be a message broadcast by the simulator q_i at line C03.

- Case 1: No correct simulator ever receives a message VALUE $(x, -)$. In this case, each correct simulator q_y is such that $values_y[x]$ remains always equal to \perp . It follows that, when q_y receives the message READ (i, x) , it sends back to q_i the message READ ANSWER (i, x, \perp) (line C21). As there are strictly more than $\frac{n}{2}$ correct simulators, q_i eventually receives the message READ ANSWER (i, x, \perp) from strictly more than $\frac{n}{2}$ different simulators, and the predicate of line C05 is then satisfied.
- Case 2: At least one correct simulator q_y receives a message VALUE (x, v) . In this case, q_y broadcasts the message VALUE (x, v) when it receives it (line C20). It follows from the broadcasts issued at this line that q_i eventually receives VALUE (x, v) from strictly more than $\frac{n}{2}$ different simulators. When this occurs, the predicate of line C06 is satisfied, and q_i exits the wait statement.

As this is true for any message READ (i, x) broadcast by the simulator q_i at line C03, it follows that q_i cannot remain block forever at lines C05-C06.

Let us finally consider the lines C12-C13. As the message VIEW (i, my_view_i) broadcast by q_i at line C12 is received by at least all the correct processes, and each of them broadcast it when it receives it for the first time, it follows that q_i receives the message VIEW (i, my_view_i) from strictly more than $\frac{n}{2}$ distinct processes, and stops waiting at line C13, which concludes the proof of the lemma. \square Lemma 1

Lemma 2. *The value returned by an invocation of propose() is a value that was proposed by a simulator.*

Proof Let us observe that (due to its definition) the set min_sigma is non-empty, and (due the first predicate of line C06) the simulator indexes y it contains are such that $values_i[y] \neq \perp$. As, for any of those y , $values_i[y]$ is set to a non- \perp value (only once) at line C20, it follows that q_i received a message VALUE (y, v_y) . Hence, the values in the variables $values_i[y]$ are values proposed by the corresponding simulators q_y . It follows that the value computed at line C18 is a value that was proposed by a simulator, which concludes the proof of the lemma. \square Lemma 2

Lemma 3. *No two invocations of `decide()` return different values.*

Proof Let us first observe that, due to the reliable broadcast of the messages `VALUE ()` (lines C01 and C20) and `VIEW ()` (lines C12 and C22), and the fact that a simulator broadcast a single message `VALUE ()`, we have:

- $(values_i[x] \neq \perp) \wedge (values_j[x] \neq \perp) \Rightarrow (values_i[x] = values_j[x]).$
- $(all_views_i[x] \neq \perp) \wedge (all_view_j[x] \neq \perp) \Rightarrow (all_views_i[x] = all_view_j[x]).$

Let us assume, by contradiction, that two simulators q_i and q_j decide different values. This means that the sets $min_σ_i$ $min_σ_j$ computed at line C17 by q_i and q_j , respectively, are different.

Since $min_σ_i$ and $min_σ_j$ are different, let us consider $z \in min_σ_i \setminus min_σ_j$ (if $min_σ_i \subsetneq min_σ_j$, swap i and j). According to the closure predicate used at line C16, as $z \notin min_σ_j$, we have $\forall y \in min_σ_j : all_views_j[y][z] = \perp$. It follows that any simulator q_y such that $y \in min_σ_j$ does not fulfill the condition of line C07 for $x = z$. Consequently, q_y received at line C05 a message `READ'ANSWER(y, z, \perp)` from a set of simulators $Q_{y,r(z)}$ of size strictly greater than $\frac{n}{2}$. Consequently when q_y executed line C03 for $x = z$, all the simulators q_k of $Q_{y,r(z)}$ verified $values_k[z] = \perp$.

When the simulator q_z stops waiting at line C02, it received messages `VALUE(z, v_z)` (where v_z is the value sent by q_z at line C01) from a set $Q_{z,w}$ of strictly more than $\frac{n}{2}$ simulators. It follows that $Q_{y,r(z)} \cap Q_{z,w} \neq \emptyset$, consequently there is a simulator q_k that sent a message `READ'ANSWER(y, z, \perp)` to q_y and a message `VALUE(z, v_z)` to q_z . Since $value_k[z]$ is never reset to \perp after being assigned, the simulator q_y necessarily executed line C03 for $x = z$ strictly before q_z stops waiting at line C02. Consequently q_y stopped waiting at line C02 before q_z executes line C03 for $x = y$. It does so after receiving messages `VALUE(y, v_y)` (where v_y is the value sent by q_y at line C01) from a set $Q_{y,w}$ of strictly more than $\frac{n}{2}$ simulators q_k , and each of these simulators then verifies $values_k = v_y$. These simulators do not send `READ'ANSWER(z, y, \perp)` messages when they receive the `READ(z, y)` message sent by q_z . Thus, it is impossible that q_z receives these messages from strictly more than $\frac{n}{2}$ processes, it consequently cannot verify the predicate of line C05. It follows that q_z executes line C12 with $my_view_z[y] = v_y \neq \perp$ and this entails that $\forall k \in \Pi : all_views_k[z] \neq \perp \Rightarrow all_views_k[z][y] \neq \perp$.

Since $z \in min_σ_i$, $all_views_i[z] \neq \perp$, $all_views_i[z][y] \neq \perp$. According to the predicate of line C16, this entails that $y \in min_σ_i$, and since the previous reasoning holds for any $y \in min_σ_j$, it shows that $min_σ_j \subseteq min_σ_i$. It follows that, when q_i executes line C17, $\forall y \in min_σ_j : all_views_i[y] \neq \perp$ and, consequently, $\forall y \in min_σ_j : all_views_i[y] = all_views_j[y]$. It entails that if $|min_σ_j| < |min_σ_i|$, then $min_σ_j$ would have been chosen by q_i at line C17, which proves that $min_σ_i = min_σ_j$ and contradicts the fact that q_i and q_j decide differently. \square Lemma 3

Lemma 4. *If no simulator crashes while executing `propose()`, then any invocation of `decide()` by a correct simulator terminates.*

Proof If no simulator crashes while executing `propose()`, it follows from Lemma 1 that every simulator q_i that invokes `propose()` broadcasts a message `VALUE (i, v_i)` at line C01 and a message `VIEW (i, my_views_i)` at line C12.

Assuming no simulator crashes while executing `propose()`, let P be the set of simulators that invoke `propose()`, and suppose that one of them, q_i , invoke `decide()` and never terminates. This can only happen if q_i waits forever for the condition of lines C15-C16 to be fulfilled. Since eventually the messages broadcast by the simulators of P are all delivered to q_i , after some finite time $\forall y \in P : all_views_i[y] \neq \perp$. Moreover, since the views broadcast by the simulators of P are built at line C08 from the messages `VALUE (-, -)` they receive, it follows that these views can contain non- \perp values only for the entries corresponding to the simulators of P (the simulators that are not in P do not sent messages `VALUE (-, -)`). Consequently, p_i eventually verifies $\forall y \in P : (all_views_i[y] \neq \perp) \wedge (\{z \in \Pi : all_views_i[y][z] \neq \perp\} \subseteq P)$. It follows that the property of lines C15-C16 is eventually true for $\sigma = P$, which contradicts the fact that q_i never terminates its `decide()` operation. \square Lemma 4

Theorem 1. *The algorithm in Figure 1 implements a safe agreement object in $\mathcal{CAM}\mathcal{P}_{n,t}[t < n/2]$.*

Proof The proof follows from Lemma 1 (Propose-Termination), Lemma 2 (Validity), Lemma 3 (Agreement), and Lemma 4 (Decide-Termination). \square *Theorem 1*

4.4 Simulation algorithm

The simulation algorithm takes as input a distributed algorithm A solving a (colorless) task in the system model $\mathcal{CAM}\mathcal{P}_{n',t}[t < n']$, and simulates it in $\mathcal{CAM}\mathcal{P}_{n,t}[t < n/2]$. Each simulator q_i , $1 \leq i \leq n$, is given a copy of the n' processes of A , and a private input vector $input_i[1..n']$, with one input per simulated processes p_j .

The simulation consists in a fair simulation by each of the n simulators q_i of the n' simulated processes p_j . To that end, each simulator manages n' threads (each simulating a process p_j), and the n threads associated with the simulation of a process p_j cooperate through safe agreement objects.

Objects shared by the simulators To produce a consistent simulation, for each simulated process p_j , the n simulators have to agree on the same sequence of the messages received by p_j . To that end, they use an array of safe agreement objects, denoted $SA[1..n', -]$, such that $SA[j, sn]$ allows them to agree on the sn -th message received by the n' threads simulating p_j at each simulator q_i .

Objects managed by each simulator q_i Each simulator manages the following data structures, with respect to each simulated process p_j .

- $input_i[j]$ contains the input of the simulated process p_j , proposed by the simulator q_i . (Simulators are allowed to propose different input vectors for the simulated processes).
- $sn_i[j]$ is the sequence number (from the simulation point of view) of the next message received by the simulated process p_j .
- $sent_i[j]$ is a sequence containing messages sent by the simulated processes to the simulated process p_j . It is assumed that the n' threads of q_i access $sent_i[j]$ in mutual exclusion (when they add messages to or withdraw messages from this sequence). The symbol \oplus is used to add messages at the end of a sequence. Sometimes $sent_i[j]$ is used as a set.
- $received_i[j]$ is a set containing the messages received by the simulated process p_j (init. \emptyset).
- $state_i[j]$ contains the current local state of the simulated process p_j . $input_i[j]$ is a part of $state_i[j]$.

It is assumed that the behavior of each simulated process p_j is described by a deterministic transition function $\delta_j()$, such that $\delta_j(state_i[j], msg)$ (a) simulates p_j until its next message reception, and (b) returns a pair. This pair is made up of the new local state of p_j plus an array $msgs[1..n']$ where $msgs[x]$ contains messages sent by p_j to the simulated process p_x .

In addition to the previous local data, each simulator q_i uses a starvation-free mutual exclusion lock, whose operations are denoted $mutex_in_i()$ and $mutex_out_i()$. This lock is used to ensure that, at any time, at most one of the n' threads of q_i access a safe agreement object. This is to guarantee that the crash of a simulator q_i entails the crash of *at most one* simulated process p_j (line 09). More precisely, if q_i crashes while executing $SA[j, sn].propose()$, it can block forever only the invocations of $SA[j, sn].decide()$, issued by the other simulators, thereby preventing the simulation of p_j from terminating.

```

(01) mutex_ini(); SA[j, 0].propose(inputi[j]); mutex_outi();
(02) inputi[j] ← SA[j, 0].decide();
(03) ⟨statei[j], msgs[1..n']⟩ ← δj(statei[j], ∅);
(04) for each x ∈ {1, ..., n'} do senti[x] ← senti[x] ⊕ msgs[x] end for;
(05) sni[j] ← 0;
(06) repeat forever
(07)   sni[j] ← sni[j] + 1;
(08)   wait ((senti[j] \ receivedi[j]) ≠ ∅);
(09)   msg ← oldest message in senti[j] \ receivedi[j];
(10)   mutex_ini(); SA[j, sni[j]].propose(msg); mutex_outi();
(11)   rec_msg ← SA[j, sni[j]].decide();
(12)   receivedi[j] ← receivedi[j] ∪ {rec_msg};
(13)   ⟨statei[j], msgs[1..n']⟩ ← δj(statei[j], rec_msg);
(14)   for each x ∈ {1, ..., n'} do senti[x] ← senti[x] ⊕ msgs[x] end for;
(15)   if (no value yet decided by pj ∧ statei[j] allows pj to decide a value v)
(16)     then the simulated process pj decides v
(17)   end if
(18) end repeat.

```

Figure 2: Thread of the simulator q_i , $1 \leq i \leq n$, simulating the process p_j , $1 \leq j \leq n'$

The simulation algorithm The algorithm describing the simulation of a process p_j by the associated thread of the simulator q_i is presented in Figure 2.

The simulators have first to agree on the same input for process p_j . To this end, they use the safe agreement object $SA[j, 0]$ (lines 01-02). Moreover, when considering all the simulated processes, it follows from the mutual exclusion lock that, whatever the number of simulated processes, a simulator q_i is engaged in at most one invocation of `propose()` at a time. Then, according to the decided input of p_j , q_i locally simulate p_j until it invokes a message reception (lines 03-04).

After this initialization, each simulator q_i enters a loop whose aim is to locally simulate p_j . To this end, q_i first determines the message that p_j will receive; this message is saved in `rec_msg` and added to `receivedi[j]` (lines 07-12). When this message has been determined, q_i simulates the behavior of p_j until its next message reception (lines 13-14). Finally, if `statei[j]` allows p_j to decide a value with respect to the simulated decision task, this value is decided (lines 15-17).

4.5 Proof of the simulation

The reader interested in a formal definition of the term *simulation* –as used here– will consult [7].

Lemma 5. *The crash of a simulator q_i entails the crash of at most one simulated process p_j .*

Proof The only places where a simulator q_i can block is during the invocation of the safe agreement operation `decide()`. Such invocations appear at line 02, and line 11. It follows from the termination property of the safe agreement objects that such an invocation can block forever the invoking process only if a simulator crashes during the invocation of the operation `propose()` on the same object. But, due to the mutual exclusion lock used at line 01 and line 10, a simulator can be engaged in at most one invocation of `propose` at a time. It follows that the crash of a simulation q_i can entail the definitive halting (crash) of at most one simulated process p_j . □ Lemma 5

Lemma 6. *The simulation of the reception of the k -th message received by a simulated process p_j , returns the same message at all simulators.*

Proof The simulation of the message receptions for a simulated process p_j , are executed at each simulator q_i at lines 08-11, and all the simulators use the same sequence of sequence numbers (line 07). It then follows

from the agreement property of the safe agreement object $SA[j, sn]$, that no two simulators obtain different messages when they invoke $SA[j, sn].decide()$, and the lemma follows. \square *Lemma 6*

Lemma 7. *For every simulated processes p_j , no two simulators return different values.*

Proof The only non-deterministic elements of the simulation are the input vectors $input_i[1..n']$ at each simulator q_i , and the reception of the simulated messages.

The lines 01-02 of the simulation force the simulators to agree on the same input value for each simulated process p_j , $1 \leq j \leq n'$. Similarly, as shown by Lemma 6, for each simulated process p_j , the lines 07-11 direct the simulators to agree on the very same sequence of messages received by p_j . It follows from the fact that the function $\delta_j()$ is deterministic, that any two simulators q_i and q_k , that execute lines 15-16 during the same “round number” $sn_i[j] = sn_k[j]$, are such that $state_i[j] = state_k[j]$, from which the lemma follows. \square *Lemma 7*

Lemma 8. *The sequences of message receptions simulated by each simulator q_i on behalf of each simulated process p_j , define a correct execution of the simulated algorithm.*

Proof To prove the correctness of the simulation, we have to show that

1. Every message that was sent by a simulated process to another simulated process (whose simulation is not blocked either), is received, and
2. The simulated messages respect a simulated physical order (i.e., no message is “received” before being “sent”).

Item 1 is satisfied because the messages sent by the simulated process p_j to the simulated process p_k are received (lines 09-11) in their sending order (as defined at line 04 and line 14). Hence, if p_k is not blocked (due to the crash of a simulator) it obtains the messages from p_j in their sending order.

For Item 2, let us define a (simulated) physical order as follows. For each simulated message m , let us consider the first time at which the reception of m was simulated (i.e., this occurs when –for the first time– a simulator terminates the invocation of $SA[-, -].decide()$ that returns m). A message that is decided has been proposed by a simulator to a safe agreement object before being decided (validity property). The sending time of a simulated message is defined as the first time at which $SA[-, -].propose(m)$ is invoked by a simulator. It follows that any simulated message is sent before being received, which concludes the lemma. \square *Lemma 8*

Lemma 9. *Each correct simulator q_i computes the decision value of at least $(n' - t)$ simulated processes.*

Proof Due to Lemma 5, and the fact that at most t simulators may crash, it follows that at most t simulated processes may be prevented from progressing. As (a) by assumption the simulated algorithm A' is t -resilient, and (b) due to Lemma 8 the simulation produces a correct simulation of A' , it follows that at least $(n' - t)$ simulated processes decide a value. \square *Lemma 9*

Theorem 2. *Let A be an algorithm solving a decision task in $\mathcal{CAMP}_{n',t}[t < n']$. The algorithm described in Figure 2 is a correct simulation of A in $\mathcal{CAMP}_{n,t}[t < n/2]$.*

Proof The theorem follows from Lemma 8 and Lemma 9. \square *Theorem 2*

5 BG(MP,B): BG in the Byzantine Asynchronous Message-Passing Model

This section presents an algorithm, denoted BG(MP,B), which implements the BG simulation in the Byzantine asynchronous message-passing model $\mathcal{BAMP}_{n,t}[t < n/3]$. To this end, an appropriate safe agreement object is first built, and then used by the simulation algorithm.

5.1 From crash failures to Byzantine behaviors

The idea is to extend the algorithm of Figure 1 to obtain an algorithm that copes with Byzantine simulators. The main issues that have to be solved are the following.

- The simulators need a mechanism to control the validity of the inputs to the safe agreement objects. (See below for the notion of a valid value.)
- The simulators must be able to check if a given simulator q_i is participating in more than one operation `propose()` at the same time (on the same or several safe agreement objects). If it is the case, q_i is faulty and its definitive stop can block forever several simulated processes. Hence, such a faulty simulator has to be ignored.

To solve these issues, each safe agreement object may no longer be considered as a separate abstraction: each new instance depends on the previous ones. This is captured in the following specification customized to the Byzantine model, and, at the operational level, in the predicate `valid()` used in the algorithm implementing the operation `propose()`.

5.2 Safe agreement in $\mathcal{BAMP}_{n,t}[t < n/3]$: definition

To cope with the previous observations, the fact that a faulty process may decide an arbitrary value, and the fact that the safe agreement objects are used to solve specific problems (a simulation in our case), the specification of the safe agreement object is reshaped as follows.

A value proposed by a process to a safe agreement object must be *valid*. At each correct simulator q_i , the validity of a value is captured by a predicate denoted $\text{valid}_i(j, v)$ where v is the value and q_j the simulator that proposed it. This predicate is made up of two parts (defined in Section 5.3 and Section 5.6, respectively). If q_j is correct, the predicate $\text{valid}_i(j, v)$ eventually returns *true* at p_i . If q_j is faulty, $\text{valid}_i(j, v)$ returns *true* at p_i only if (a) the value v could have been proposed by a correct simulator and (b) to q_i 's knowledge, q_j does not participate concurrently in several invocations of `propose()`.

- **Validity.** If a correct simulator q_i decides the value v , there is a correct simulator q_j such that $\text{valid}_j(-, v)$. (v was validated by a correct simulator.)
- **Agreement.** No two correct simulators decide distinct values.
- **Propose-Termination.** Any invocation of `propose()` by a correct simulator terminates.
- **Decide-Termination.** The invocations by all the correct simulators of `decide()` on all the safe agreement objects terminate, except for at most t safe agreement objects.

5.3 Safe agreement in $\mathcal{BAMP}_{n,t}[t < n/3]$: algorithm

The local variables $\text{values}_i[1..n]$, $\text{my_view}_i[1..n]$, $\text{all_views}_i[1..n]$, and the algorithm implementing the operation `decide()` are the same as in Figure 1 (lines C15-C19). The new algorithm implementing the operation `propose()`, and the processing of the associated messages, are described in Figure 3 and Figure 4.

This implementation uses an additional local array $\text{answers}_i[1..n][1..n][1..n]$, all entries of which are initialized to “?”. The meaning of “ $\text{answers}_i[k][j][x] = v$ ” (where v is a proposed value or \perp) is the following: to the knowledge of q_i , the simulator q_k answered value v when it received the message `READ(j, x)` sent by q_j . (A simulator q_j broadcasts such a message when it needs to know the value proposed by the simulator q_x ; \perp means that q_k does not know this value yet.) This means that, from q_i 's point of view, the value proposed by q_x , as known by q_k when it received the request by q_j , is v .

Lemma 10. *Any two sets of simulators Q_1 and Q_2 of more than $\frac{n+t}{2}$ elements have at least one correct simulator in their intersection.*

Proof As we consider integers, “strictly more than $\frac{n+t}{2}$ ” is equivalent to “at least $\lfloor \frac{n+t}{2} \rfloor + 1$ ”.

- $Q_1 \cup Q_2 \subseteq \{p_1, \dots, p_n\}$. Hence, $|Q_1 \cup Q_2| \leq n$.
- $|Q_1 \cap Q_2| = |Q_1| + |Q_2| - |Q_1 \cup Q_2| \geq |Q_1| + |Q_2| - n \geq 2(\lfloor \frac{n+t}{2} \rfloor + 1) - n > 2(\frac{n+t}{2}) - n = t$.
Hence, $|Q_1 \cap Q_2| \geq t + 1$. It follows that $Q_1 \cap Q_2$ contains at least one correct simulator.

□ *Lemma 10*

The fact that, despite Byzantine processes, the intersection of any two simulator sets of size greater than $\frac{n+t}{2}$ have at least one correct simulator in common, is used in many places in the algorithm. This property will be used in the proof to show that the local views of the correct processes are mutually consistent.

The operation propose() The client side of the algorithm implementing the operation propose() is described in Figure 3; its server side is described in Figure 4. The client side algorithm is very close to the one of the crash failure case (Figure 1). They differ in two points.

- The message tags VALUE and VIEW (used at lines C02, C06, and C13 in Figure 1) are replaced in Figure 3 by the tags VALUE’ACK and VIEW’ACK, respectively. The role of these message tags is explained below.
- The predicate of line B05 is replaced by the predicate $|\{k : answers_i[k][i][x] = \perp\}| > \frac{n+t}{2}$. This predicate states that more than $\frac{n+t}{2}$ simulators answered \perp to the request message READ(i, x) broadcast by q_i , (i.e., they did not know the value proposed by q_x when they received the read request).

```

operation propose( $v_i$ ) is
(B01) broadcast VALUE( $i, v_i$ );
(B02) wait (VALUE’ACK( $i, v_i$ ) received from  $> \frac{n+t}{2}$  different simulators);
(B03) for each  $x \in [1..n]$  do broadcast READ( $i, x$ ) end for;
(B04) for each  $x \in [1..n]$  do
(B05)   wait ( $|\{k : answers_i[k][i][x] = \perp\}| > \frac{n+t}{2} \vee$ 
(B06)      $(\exists w : \text{VALUE’ACK}(x, w) \text{ received from } > \frac{n+t}{2} \text{ different simulators}))$ ;
(B07)   if (predicate of line B06 satisfied)
(B08)     then  $my\_view_i[x] \leftarrow w$ 
(B09)     else  $my\_view_i[x] \leftarrow \perp$ 
(B10)   end if
(B11) end for;
(B12) broadcast VIEW( $i, my\_view_i$ );
(B13) wait (VIEW’ACK( $i, my\_view_i$ ) received from  $> \frac{n+t}{2}$  different simulators);
(B14) return().

operation decide() is
(C15) wait ( $\exists$  a non-empty set  $\sigma \subseteq \Pi$ :
(C16)  $\forall y \in \sigma : [(all\_views_i[y] \neq \perp) \wedge (\forall z \in \Pi : (all\_views_i[y][z] \neq \perp) \Rightarrow (z \in \sigma))]$ );
(C17) let  $min\_sigma_i$  be the set  $\sigma$  of smallest size;
(C18) let  $res$  be  $\min(\{values_i[y] : y \in min\_sigma_i\})$ ;
(C19) return( $res$ ).

```

Figure 3: Safe agreement object in $\mathcal{BAMP}_{n,t}[t < n/3]$: operation propose() of simulator q_i

Messages VALUE(), VALUE’VALID(), VALUE’WITNESS() **and** VALUE’ACK() When a simulator q_i invokes the operation propose(v_i), it first broadcasts the message VALUE(i, v_i), and waits for $\frac{n+t}{2}$ acknowledgments (messages VALUE’ACK(i, v_i), lines B01-B02). Then, as in the crash failure case (Figure 1), it builds its local view of the values proposed to the safe agreement object (lines B03-B11). Finally, it sends its local view to all other simulators (lines B12-B13).

when the message VALUE (j, v) **is received from** q_j **for the first time:**
(B15) **wait** ($\text{valid}_i(j, v)$); **broadcast** VALUE'VALID (j, v).

when the message VALUE'VALID (j, v) **is received:**
(B16) **if** ((VALUE'VALID (j, v) received from $> \frac{n+t}{2}$ different simulators) \wedge (VALUE'WITNESS ($j, -$) never broadcast))
(B17) **then** broadcast VALUE'WITNESS (j, v) **end if**.

when the message VALUE'WITNESS (j, v) **is received:**
(B18) **if** ((VALUE'WITNESS (j, v) received from $t + 1$ different simulators) \wedge (VALUE'WITNESS (j, v) never broadcast))
(B19) **then** broadcast VALUE'WITNESS (j, v)
(B20) **end if**;
(B21) **if** (VALUE'WITNESS (j, v) received from $> \frac{n+t}{2}$ different simulators)
(B22) **then** $\text{values}_i[j] \leftarrow v$; **broadcast** VALUE'ACK (j, v)
(B23) **end if**.

when the message READ (j, x) **is received from** q_j **for the first time:**
(B24) **wait** (VALUE'ACK (j, v) received from $> \frac{n+t}{2}$ different simulators);
(B25) $\text{values}_i[j] \leftarrow v$; **broadcast** VALUE'ACK (j, v);
(B26) **broadcast** READ'ANSWER ($j, x, \text{values}_i[x]$).

when the message READ'ANSWER (j, x, v) **is received from** q_k **for the first time:**
(B27) **if** (READ'ANSWER'WITNESS ($k, j, x, -$) never broadcast) **then** broadcast READ'ANSWER'WITNESS (k, j, x, v) **end if**.

when the message READ'ANSWER'WITNESS (k, j, x, v) **is received:**
(B28) **if** ((READ'ANSWER'WITNESS (k, j, x, v) received from $t + 1$ different simulators)
(B29) \wedge (READ'ANSWER'WITNESS (k, j, x, v) never broadcast))
(B30) **then** broadcast READ'ANSWER'WITNESS (k, j, x, v)
(B31) **end if**;
(B32) **if** (READ'ANSWER'WITNESS (k, j, x, v) received from $> \frac{n+t}{2}$ different simulators)
(B33) **then** $\text{answers}_i[k][j][x] \leftarrow v$
(B34) **end if**.

when the message VIEW (j, view) **is received from** q_j **for the first time:**
(B35) **if** ((VIEW'WITNESS ($j, -$) never broadcast) \wedge ($\text{view}[j] \neq \perp$))
(B36) **then for** $x \in [1..n]$ **do**
(B37) **if** ($\text{view}[x] \neq \perp$)
(B38) **then wait** (VALUE'ACK ($x, \text{view}[x]$) received from $> \frac{n+t}{2}$ different simulators)
(B39) **else wait** ($|\{k : \text{answers}_i[k][j][x] = \perp\}| > \frac{n+t}{2}$)
(B40) **end if**
(B41) **end for**;
(B42) **broadcast** VIEW'WITNESS (j, view)
(B43) **end if**.

when the message VIEW'WITNESS (j, view) **is received:**
(B44) **if** ((VIEW'WITNESS (j, view) received from $t + 1$ different simulators) \wedge (VIEW'WITNESS (j, view) never broadcast))
(B45) **then** broadcast VIEW'WITNESS (j, view)
(B46) **end if**;
(B47) **if** (VIEW'WITNESS (j, view) received from $> \frac{n+t}{2}$ different simulators)
(B48) **then** $\text{all_views}_i[j] \leftarrow \text{view}$; **send** VIEW'ACK (j, view) **to** q_j
(B49) **end if**.

Figure 4: Safe agreement object in $\mathcal{BAMP}_{n,t}[t < n/3]$: server side of simulator q_i

On its server side, when a simulator q_i receives a message VALUE (j, v), it first checks if this message is valid (line B15). If the message is valid, q_i broadcasts (echoes) the message VALUE'VALID (j, v) to inform

the other simulators that it agrees to take into account the pair (j, v) (line B15).

When the simulator p_i has received the message VALUE'VALID (j, v) from more than $\frac{n+t}{2}$ simulators, it broadcasts the message VALUE'WITNESS (j, v) to inform the other processes that at least $\frac{n+t}{2} - t = \frac{n-t}{2} \geq t + 1$ correct simulators, have validated the pair (j, v) .

When q_i has received the message VALUE'WITNESS (j, v) from $(t + 1)$ simulators (i.e., from at least one correct simulator) it broadcasts this message, if not yet done (lines B18-B20). This is to prevent invocations of propose() from blocking forever (while waiting VALUE'ACK (j, v) messages at line B02, B06, B24 or B38), because not enough VALUE'WITNESS (j, v) messages have been broadcast². Then, if q_i has received the message VALUE'WITNESS (j, v) from more than $\frac{n+t}{2}$ simulators, it takes v into account (writes it into $values_i[j]$) and sends an acknowledgment to q_j (lines B21-B23). The corresponding message VALUE'ACK (j, v) broadcast by q_i will also inform the other simulators that q_i took into account the value v proposed by q_j . Hence, this message will help q_j progress at line B02, and all correct simulators progress at line B06.

First part of the predicate $valid_i(j, v)$ As already indicated, the aim of this predicate is to help a simulator q_i detect if the value v proposed by the simulator q_j is valid. It is always satisfied when q_j is correct, and it can return *true* or *false* when q_j is faulty. It is made up of two sub-predicates $P1$ and $P2$.

- The first sub-predicate $P1$ checks if, for the messages VALUE $(j, -)$ (from q_j) and VALUE'VALID $(j, -)$ (from more than $t + 1$ different simulators) that q_i has received for other safe agreement objects, q_i has also received the associated messages VIEW'WITNESS $(j, -)$ from at least $(n - t)$ different simulators. This allows q_i to check if the simulator q_j is not simultaneously participating in other invocations of propose() on other safe agreement objects.
- The aim of the second sub-predicate $P2$ (defined in Section 5.6 and used in the simulation) is to allow the simulators to check that the simulation is consistent. As the present section considers safe agreement objects independently from its use in the simulation, we consider, for now, that $P2$ is always satisfied.

If the full predicate $valid_i(j, v)$ is never satisfied, q_i will, collectively with the other correct simulators, prevent the faulty simulator q_j from progressing with respect to the corresponding safe agreement object.

Messages READ(), READ'ANSWER() and READ'ANSWER'WITNESS() After the value v_i it proposes to the safe agreement object has been taken into account by $\frac{n+t}{2}$ simulators, q_i builds a local view of all the values proposed (array $my_view_i[1..n]$). To this end, as in the crash failure model, q_i sends to each simulator q_x the customized message READ (i, x) (line B03). Its behavior is then similar to the one of the crash failure model (line B04-B11), where the new predicate $|\{k : answers_i[k][i][x] = \perp\}| > \frac{n+t}{2}$ is now used at line B05.

When q_i receives the message READ (j, x) from the simulator q_j , it first waits until it knows that the value proposed by q_j is known by more than $\frac{n+t}{2}$ simulators (line B24). This is to check that q_j broadcast its proposed value before reading the other simulator values used to build its own view. When this occurs, q_i answers the message READ (j, x) by broadcasting the message READ'ANSWER $(j, x, values_i[x])$ to inform all the simulators on what it currently knows on the value proposed by q_x (line B26). (Let us remind that, in the crash failure model, q_i was sending this message only to q_j .)

When it receives the message READ'ANSWER (j, x, v) from a simulator q_k , if not yet done, q_i broadcasts the message READ'ANSWER'WITNESS (k, j, x, v) . The lines B27-B31 implement a reliable broadcast [8], i.e., the message READ'ANSWER'WITNESS (k, j, x, v) is received by all correct processes or none of them, and is always received if the sender is correct. The reliable reception of this message entails the assignment of $answers_i[k, j, x]$ to v (line B33).

²A similar mechanism is used in [8] to ensure that the proposed reliable broadcast abstraction guarantees that a message is received by all or none of the correct processes.

Messages VIEW(), VIEW'WITNESS() and VIEW'ACK() Finally, as in Figure 1, the simulator q_i broadcasts its local view of proposed values to all simulators, waits until more than $\frac{n+t}{2}$ of them sent back an acknowledgment, and returns from the invocation of propose() (lines B12-B14).

When q_i receives for the first time the message VIEW $(j, view)$, it realizes an enriched reliable broadcast whose aim is to assign $view$ to $all_view_i[j]$. Let us first observe that if $view[j] = \perp$, then q_j is Byzantine. If it has not yet broadcast VIEW'WITNESS $(j, view)$ and if $view[j] \neq \perp$ (line B35), q_i first checks if all the values in $view[1..n]$ are consistent. From its point of view, this means that, for each simulator q_x , (a) if $view[x] = v$, it must receive messages VALUE'ACK (x, v) from more than $\frac{n+t}{2}$ simulators, and (b) if $view[x] = \perp$, the same predicate as in line B05 must become satisfied. This consistency check is realized by the lines B36-B41.

Finally, when q_i receives a message VIEW'WITNESS $(j, view)$, it does the following. First, if it has received this message from at least one correct simulator, and has not yet broadcast it, q_i does it (lines B44-B46). This part of the reliable broadcast is to prevent the correct simulators from blocking forever. Then, if it has received VIEW'WITNESS $(j, view)$ from more than $\frac{n+t}{2}$ simulators and has not yet assigned a value to $all_view_i[j]$, q_i does it and sends to q_j the acknowledgment message VIEW'ACK $(j, view)$ to inform q_j that it knows its view (lines B47-B49).

5.4 A communication pattern

When considering the algorithm of Figure 4, it appears that the processing of the messages VALUE'WITNESS () (lines B18-B23), READ'ANSWER'WITNESS () (lines B28-B34), and VIEW'WITNESS () (lines B44-B49), follow the same generic pattern. This pattern, inspired from [8] and where WITNESS is used as message tag, is described in Figure 5.

```

when WITNESS ( $m$ ) is received:
(GP01) if (WITNESS ( $m$ ) received from  $t + 1$  different simulators
(GP02)  $\wedge$  WITNESS ( $m$ ) never broadcast)
(GP03) then broadcast WITNESS ( $m$ )
(GP04) end if;
(GP05) if (WITNESS ( $m$ ) received from  $> \frac{n+t}{2}$  different simulators)
(GP06) then execute statement  $A$ 
(GP07) end if.

```

Figure 5: Generic communication pattern in $\mathcal{BAMP}_{n,t}[t < n/3]$

Theorem 3. (i) *If a correct simulator executes action A , all correct simulators do it.*

(ii) *If $(t + 1)$ correct simulators execute broadcast WITNESS(m), all correct simulators execute action A .*

Proof Proof of (i). Let p_i be a correct process that executes A . It follows from line GP05 that it has received the message WITNESS (m) from more than $\frac{n+t}{2}$ different simulators. As $n > 3t$, $\lfloor \frac{n+t}{2} \rfloor + 1 \geq 2t + 1$, p_i received the message WITNESS (m) from at least $(t + 1)$ correct simulators. It then follows from lines GP01-GP02 that all correct simulators broadcast WITNESS (m) and, consequently, each correct simulator receives WITNESS (m) from at least $(n - t)$ simulators. The proof follows from $n - t > \frac{n+t}{2}$.

Proof of (ii). If $(t + 1)$ correct simulators broadcast WITNESS (m), the predicate of line GP01 is eventually satisfied at every correct simulator. As $n - t > \frac{n+t}{2}$, it follows that the predicate of line GP05 will also be satisfied at each correct simulator, which concludes the proof. \square Theorem 3

5.5 Safe agreement object in $\mathcal{BAMP}_{n,t}[t < n/3]$: proof

This section proves that the algorithm presented in Figures 3 and 4 implements a safe agreement object in the presence of Byzantine simulators, i.e., any of its runs in $\mathcal{BAMP}_{n,t}[t < n/3]$ satisfies the validity, agreement, and termination properties that define this object.

Propose-termination

Lemma 11. *Let q_i be a correct simulator. If the predicate $\text{valid}_j(i, v_i)$ eventually becomes satisfied at the correct simulators q_j , then the invocation of $\text{propose}(v_i)$ by q_i terminates.*

Proof A correct simulator q_i can be blocked forever in a wait statement (1) at line B02, (2) at lines B05-B06, or (3) at line B13. We show that, if the predicate $\text{valid}_j(i, v_i)$ is eventually satisfied at the correct simulators q_j , p_i cannot block forever in the invocation of $\text{propose}(v_i)$.

- wait instruction at line B02.

Simulator q_i first broadcasts the message $\text{VALUE}(i, v_i)$ (line B01), then waits for $\text{VALUE}'\text{ACK}$ messages from more than $\frac{n+t}{2}$ different simulators. When a correct simulator q_j receives $\text{VALUE}(i, v_i)$ for the first time, it waits until $\text{valid}_j(i, v_i)$ becomes satisfied. By assumption, this happens. Simulator q_j then broadcasts $\text{VALUE}'\text{VALID}(i, v_i)$. It follows that each of the at least $(n - t)$ correct simulators broadcasts the message $\text{VALUE}'\text{VALID}(i, v_i)$.

As $n - t > \frac{n+t}{2}$, it follows that each correct simulator q_j receives the message $\text{VALUE}'\text{VALID}(i, v_i)$ from more than $\frac{n+t}{2}$ simulators and broadcasts the message $\text{VALUE}'\text{WITNESS}(i, v_i)$.

According to Theorem 3, q_j updates $\text{values}_j[i]$ with v_i , and broadcasts $\text{VALUE}'\text{ACK}(i, v_i)$ (lines B21-B23). The correct simulator q_i will then receive the message $\text{VALUE}'\text{ACK}(i, v_i)$ from at least $n - t > \frac{n+t}{2}$ simulators. Hence, it cannot block forever at line B02.

- wait instruction at lines B05-B06.

In this waiting statement, q_i waits until either $|\{k : \text{answers}_i[k][i][j] = \perp\}| > \frac{n+t}{2}$ becomes true, or until it receives $\text{VALUE}'\text{ACK}(j, w)$ from more than $\frac{n+t}{2}$ different simulators.

- If q_j is a correct simulator that invoked $\text{propose}(j, w)$, the reasoning is the same as above. Consequently, q_i will receive $\text{VALUE}'\text{ACK}(j, w)$ from at least $n - t > \frac{n+t}{2}$ different simulators.
- If q_j is faulty or never invokes $\text{propose}(j, w)$, q_i may never receive $\text{VALUE}'\text{ACK}(j, w)$ from more than $\frac{n+t}{2}$ different simulators. We will show that, in this case, the wait predicate $|\{k : \text{answers}_i[k][i][j] = \perp\}| > \frac{n+t}{2}$ eventually becomes true.

We first show that, if a correct simulator receives $\text{VALUE}'\text{ACK}(j, w)$ from more than $\frac{n+t}{2}$ different simulators, then all correct simulators do receive $\text{VALUE}'\text{ACK}(j, w)$ from more than $\frac{n+t}{2}$ different simulators. If a correct simulator receives $\text{VALUE}'\text{ACK}(j, w)$ from more than $\frac{n+t}{2}$ different simulators, at least $(t + 1)$ correct simulators broadcast it. Every correct simulator will then receive the message $\text{VALUE}'\text{ACK}(j, w)$ from at least $(t + 1)$ different simulators and, if not already done, broadcasts it (lines B24-B25). All correct simulators will then receive the message $\text{VALUE}'\text{ACK}(j, w)$ from at least $n - t > \frac{n+t}{2}$ different simulators.

According to the previous observation, let us consider the case in which no correct simulator ever receives the message $\text{VALUE}'\text{ACK}(j, w)$ from more than $\frac{n+t}{2}$ different simulators. A correct simulator q_k assigns a non- \perp value to $\text{values}_k[j]$ only if it receives $\text{VALUE}'\text{WITNESS}(j, w)$ from more than $\frac{n+t}{2}$ different simulators (line B22), or if it receives $\text{VALUE}'\text{ACK}(j, w)$ from more than $\frac{n+t}{2}$ different simulators (line B25). If a correct simulator receives $\text{VALUE}'\text{WITNESS}(j, w)$ from more than $\frac{n+t}{2}$ different simulators, according to Theorem 3, all correct simulators receive $\text{VALUE}'\text{WITNESS}(j, w)$ from more than $\frac{n+t}{2}$ different simulators, and broadcast the message $\text{VALUE}'\text{ACK}(j, w)$. Because no correct simulator ever receives $\text{VALUE}'\text{ACK}(j, v_j)$ messages from more than $\frac{n+t}{2}$ different simulators, no correct simulator q_k will ever assign a non- \perp value to $\text{values}_k[j]$ (line B22).

When a correct simulator receives a $\text{READ}(i, j)$ message from q_i , it waits until it has received $\text{VALUE}'\text{ACK}(i, v_i)$ messages from more than $\frac{n+t}{2}$ different simulators (line B24). The reasoning above (first item) shows that this will eventually become true.

Every correct simulator q_k will then broadcast $\text{READ'ANSWER}(i, j, \perp)$. This will cause all correct simulators to broadcast messages $\text{READ'ANSWER'WITNESS}(k, i, j, \perp)$, which will be received by the simulator q_i . This will then assign \perp to $\text{answers}_i[k][i][j]$ for at least $n - t > \frac{n+t}{2}$ different values of k . Consequently, it will not remain blocked at lines B05-B06.

- wait instruction at line B13.

As simulator q_i broadcasts its view with a message $\text{VIEW}(i, \text{view})$ (line B12), every correct simulator checks if this view is consistent when it receives it (lines B36-B41). Let us first consider the entries $\text{view}[j]$ such that $\text{view}[j] = w \neq \perp$. This means that q_i has received $\text{VALUE'ACK}(j, w)$ from more than $\frac{n+t}{2}$ different simulators. All the correct simulators then receive the same message from a sufficient number of different simulators and do not remain blocked at line B38 (Theorem 3).

Let us now consider the entries $\text{view}[j]$ such that $\text{view}[j] = \perp$. Simulator q_i assigned \perp to $\text{view}[j]$ because it received $\text{READ'ANSWER'WITNESS}(k, i, j, \perp)$ from more than $\frac{n+t}{2}$ different simulators (lines B32-33). According to Theorem 3, all the correct simulators q_x will also receive $\text{READ'ANSWER'WITNESS}(k, i, j, \perp)$ from more than $\frac{n+t}{2}$ different simulators, and will assign \perp to $\text{answers}_x[k][i][j]$. They will thus not remain blocked at line B39.

All the correct simulators will then broadcast the message $\text{VIEW'WITNESS}(i, \text{view})$ (line B42). By Theorem 3, they will all send $\text{VIEW'ACK}(i, \text{view})$ to q_i . This will allow q_i to terminate its invocation of $\text{propose}(i, v_i)$, which concludes the proof of the lemma.

□ Lemma 11

Lemma 12. *Let v_1, \dots, v_x, \dots be the values proposed by a correct simulator q_i to a sequence of safe agreement objects. If q_i does not invoke $\text{propose}()$ operations concurrently and $\text{valid}_j(i, v_x)$ is eventually satisfied at every correct simulator q_j , then $\text{valid}_j(i, v_{x+1})$ is also eventually satisfied at q_j .*

Proof We consider here that the sub-predicate $P2$ is always satisfied, and thus consider only the sub-predicate $P1$. Let us recall that $P1$ states that, for every message $\text{VALUE}(i, -)$ that q_j received from q_i , and for every message $\text{VALUE'VALID}(i, -)$ that q_j received from at least $t + 1$ different simulators, it has also received the corresponding messages $\text{VIEW'WITNESS}(i, -)$.

By hypothesis, $\text{valid}_j(i, v_x)$ is eventually satisfied at the correct simulator q_j . Once q_i broadcasts the message $\text{VALUE}(i, v_x)$, q_j only needs to receive the corresponding $\text{VIEW'WITNESS}(i, \text{view})$ for $P1$ to be satisfied. By Lemma 11, q_i terminates its invocation of $\text{propose}(i, v_x)$, from which we conclude that it received $\text{VIEW'ACK}(i, \text{view})$ from more than $\frac{n+t}{2}$ different simulators (line B13). A correct simulator sends such a message only if it has received $\text{VIEW'WITNESS}(i, \text{view})$ from more than $\frac{n+t}{2}$ different simulators (lines B47-B48). According to Theorem 3, all the correct simulators also broadcast it (lines B44-B45). The correct simulator q_j then receives them from more than $\frac{n+t}{2}$ different simulators. The predicate $\text{valid}_j(i, v_{x+1})$ is then eventually satisfied at q_j .

□ Lemma 12

Decide-termination

Lemma 13. *If a correct simulator terminates its invocation of $\text{decide}()$, then all correct simulators terminate their invocation of $\text{decide}()$.*

Proof Suppose, by way of contradiction, that the invocation of $\text{decide}()$ by a correct simulator q_i terminates, and that the invocation of $\text{decide}()$ by another correct simulator q_j does not.

The invocation of $\text{decide}()$ by q_i can terminate only if the predicate at lines C15-C16 is satisfied. Let q_k be any simulator in the set σ defined at line C15. We show that $\text{all_views}_i[k] = \text{view}$ implies that we eventually have $\text{all_views}_j[k] = \text{view}$, and thus that q_j must decide.

Simulator q_i assigns view to $\text{all_views}_i[k]$ at line B48. This can happen only because q_i received $\text{VIEW'WITNESS}(k, \text{view})$ messages from more than $\frac{n+t}{2}$ different simulators. According to Theorem 3, q_j

eventually receives enough $\text{VIEW}'\text{WITNESS}(k, \text{view})$ messages and also assigns view to $\text{all_views}_j[k]$. Simulator q_j will then also have to decide. \square Lemma 13

Lemma 14. *The invocations of $\text{decide}()$ by all the correct simulators on all the safe agreement objects terminate, except for at most t safe agreement objects.*

Proof Suppose, by way of contradiction, that there are $t + 1$ safe agreement objects such that at least one correct simulator never terminates its invocation of $\text{decide}()$. By Lemma 13, there must be $(t + 1)$ different safe agreement objects in which no correct simulator terminates its invocations of $\text{decide}()$.

The invocation of the $\text{decide}()$ operation by a correct simulator q_i on a safe agreement object can only be blocked at lines C15-C16, if the corresponding predicate is never satisfied. This can happen if (1) there is no simulator q_j such that $\text{all_views}_i[j] \neq \perp$ or, (2) for every non-empty set of simulators σ , there are two simulators $q_y \in \sigma$ and q_z such that $\text{all_views}_i[y][z] \neq \perp \wedge \text{all_views}_i[z] = \perp$. Because a correct simulator q_i invokes $\text{propose}()$ before invoking $\text{decide}()$, case (1) cannot happen; we always have $\text{all_views}_i[i] \neq \perp$. We then consider case (2).

Case (2) can happen if q_z starts an invocation of $\text{propose}()$ and communicates its proposed value to other processes, but does not terminate its invocation by communicating its view. Because there are at most t faulty simulators, by the pigeonhole principle, there must be a faulty simulator q_z that prevents q_i from deciding on two different safe agreement objects.

A correct simulator q_k broadcasts a $\text{VALUE}'\text{VALID}(z, -)$ after receiving a $\text{VALUE}(z, -)$ message only if the predicate $\text{valid}_k(z, -)$ is satisfied (line B15). Due to the predicate $\text{valid}_k(z, -)$, this is true only if q_k received $\text{VIEW}'\text{WITNESS}(z, -)$ messages from at least $(n - t)$ different simulators, each of these messages corresponding to the all the $\text{VALUE}(z, -)$ and $\text{VALUE}'\text{VALID}(z, -)$ messages that it has previously received (see the definition of the predicate $P1$ of $\text{valid}_k(z, -)$).

Let $\text{propose}(v_1)$ be the invocation of $\text{propose}()$ by q_z on the first safe agreement object on which q_i is blocked, and $\text{propose}(v_2)$ the one on the second safe agreement object on which q_i is blocked. Because there is a simulator $q_y \in \sigma$ such that $\text{all_views}_i[y] \neq \perp$ in the two invocations of $\text{decide}()$ by q_i , in both cases, more than $\frac{n+t}{2}$ different simulators have broadcast a $\text{VIEW}'\text{WITNESS}(y, -)$ message (line B48). Both sets include correct simulators. They must then have received $\text{VALUE}'\text{ACK}(z, v_1)$ and $\text{VALUE}'\text{ACK}(z, v_2)$ from more than $\frac{n+t}{2}$ different simulators (line B38). Again, both sets include correct simulators that must have received $\text{VALUE}'\text{WITNESS}(z, v_1)$ and $\text{VALUE}'\text{WITNESS}(z, v_2)$ from more than $\frac{n+t}{2}$ different simulators (line B21).

In order to broadcast a $\text{VALUE}'\text{WITNESS}(z, -)$ message, a correct simulator must either (a) receive $\text{VALUE}'\text{WITNESS}(z, -)$ messages from at least $t+1$ different simulators (line B18), or (b) receive $\text{VALUE}'\text{VALID}(z, -)$ messages from more than $\frac{n+t}{2}$ different simulators (line B16). The first correct simulator that broadcasts a $\text{VALUE}'\text{WITNESS}(z, -)$ message must then have received $\text{VALUE}'\text{VALID}(z, -)$ messages from more than $\frac{n+t}{2}$ different simulators.

According to Lemma 10, there is a least one correct simulator q_ℓ that broadcasts both $\text{VALUE}'\text{VALID}(z, -)$ messages (line B15). In order to do so, the predicate $\text{valid}_\ell(z, v_2)$ must have been verified at the time that q_ℓ broadcast the $\text{VALUE}'\text{VALID}(z, v_2)$ message. It must then have received the $\text{VIEW}'\text{WITNESS}(z, \text{view})$ messages that correspond to v_1 from more than $\frac{n+t}{2}$ different simulators. According to Theorem 3, q_i must then also have received these messages from more than $\frac{n+t}{2}$ different simulators and assigned view to $\text{all_views}_i[z]$ (line B48) in the instance that corresponds to the invocation of $\text{propose}(v_1)$ by q_z , a contradiction that concludes the proof of the lemma. \square Lemma 14

Agreement

Lemma 15. *For any simulator q_x and any correct simulator q_i , if q_i assigns a non- \perp value v to $\text{values}_i[x]$, then (1) no value $v' \neq v$ is ever assigned to $\text{values}_j[x]$ by a correct simulator q_j and (2) each such correct simulator q_j eventually assigns v to $\text{values}_j[x]$.*

Proof Let q_k be the first simulator that assigns v to $values_k[x]$. Since q_k executes line B22, it received strictly more than $\frac{n+t}{2}$ VALUE'WITNESS (x, v) messages from different simulators. At least $t + 1$ correct simulators consequently sent this message to all processes at line B17 or at line B19. By Theorem 3, every correct simulator q_j consequently eventually receives such a message from each correct simulator and assigns v to $values_j[x]$.

Suppose that there exists a value $v' \neq v$ such that there is a correct simulator q_ℓ that assigns v' to $values_\ell[x]$. Suppose that q_ℓ is the first process to do so. It follows that q_ℓ received VALUE'WITNESS (x, v') messages from strictly more than $\frac{n+t}{2}$ different processes (line B21 or line B24).

Consider the first correct simulator that broadcasts a VALUE'WITNESS (x, v') message. In order to do so, it must have received VALUE'VALID (x, v') messages from strictly more than $\frac{n+t}{2}$ different processes (lines B16-B17). However, the first correct simulator that broadcasts a VALUE'WITNESS (x, v) message must also have received VALUE'VALID (x, v) messages from strictly more than $\frac{n+t}{2}$ different processes. There must then be a correct simulator that sent both VALUE'VALID $(x, -)$ messages. The only place a correct simulator can send a VALUE'VALID $(x, -)$ message is at Line 15 and it does so only once for each simulator q_x , a contradiction which concludes the proof of the lemma. \square Lemma 15

Lemma 16. *For any simulators q_k, q_ℓ, q_x and any correct simulator q_i , if q_i assigns a non- \perp value v to $answers_i[\ell][k][x]$, then (1) no value $v' \neq v$ is ever assigned to $answers_j[\ell][k][x]$ by a correct simulator q_j and (2) each such correct simulator q_j eventually assigns v to $answers_j[\ell][k][x]$.*

Proof The proof is the same as for Lemma 15. \square Lemma 16

Lemma 17. *For any simulator q_x and any correct simulator q_i , if q_i assigns a non- \perp value $view$ to $all_views_i[x]$, then (1) no value $view' \neq view$ is ever assigned to $all_views_j[x]$ by a correct simulator q_j and (2) each such correct simulator q_j eventually assigns $view$ to $all_views_j[x]$.*

Proof The proof is the same as for Lemma 15. \square Lemma 17

Lemma 18. *No two invocations of `decide()` return different values.*

Proof Let us recall that the algorithm implementing the operation `decide()` is described at lines C15-C19. Let q_i and q_j be two correct simulators. According to Lemmas 15-17, we have:

- $(values_i[x] \neq \perp) \wedge (values_j[x] \neq \perp) \Rightarrow (values_i[x] = values_j[x])$.
- $(answers_i[\ell][k][x] \neq ? \wedge answers_j[\ell][k][x] \neq ?) \Rightarrow (answers_i[\ell][k][x] = answers_j[\ell][k][x])$.
- $(all_views_i[x] \neq \perp \wedge all_view_j[x] \neq \perp) \Rightarrow (all_views_i[x] = all_view_j[x])$.

Let us assume, by contradiction, that q_i and q_j decide different values. This means that the sets min_sigma_i and min_sigma_j computed at line C17 by q_i and q_j , respectively, are different.

Since min_sigma_i and min_sigma_j are different, let us consider $z \in min_sigma_i \setminus min_sigma_j$ (if $min_sigma_i \subsetneq min_sigma_j$, swap i and j). According to the closure predicate used at line C16, as $z \notin min_sigma_j$, we have $\forall y \in min_sigma_j : all_views_j[y][z] = \perp$.

It follows that q_j received VIEW'WITNESS $(y, all_view_j[y])$ messages (with $all_view_j[y][z] = \perp$) from a set of simulators $Q_{j,vw}$ of size strictly larger than $\frac{n+t}{2}$ (the subscript vw stands for “view witness”). The correct simulators of $Q_{j,vw}$ sent these messages after checking at line B39 that a set $Q_{j,vw,r}$ of strictly more than $\frac{n+t}{2}$ reliably broadcast (thanks to the mechanism of lines B26 to B33) a READ'ANSWER (y, z, \perp) message. The correct simulators of $Q_{j,vw,r}$ sent these messages at line B26 after they received VALUE'ACK (y, v_y) messages from a set $Q_{y,w}$ of strictly more than $\frac{n+t}{2}$ simulators (the subscript w stands for “witness”). Each correct simulator q_k of $Q_{y,w}$ had $values_k[y] = v_y$ when it sent this message and it happens strictly before the first correct simulator sends a READ'ANSWER (y, z, \perp) message.

Since $z \in \min_ \sigma_i$, the correct simulator q_i received VIEW'WITNESS $(z, all_view_i[z])$ messages from a set $Q_{i,vw}$ of strictly more than $\frac{n+t}{2}$ simulators. The correct simulators of $Q_{i,vw}$ sent these messages after the check of the values at lines B38-B39.

Suppose that some of them verified the predicate of line B39 for $x = y$. It entails that a set $Q_{i,vw,r}$ of strictly more than $\frac{n+t}{2}$ simulators reliably broadcast a READ'ANSWER (z, y, \perp) . The correct simulators of $Q_{i,vw,r}$ sent this message after receiving at line B24 VALUE'ACK (z, v_z) messages from a set $Q_{z,w}$ of strictly more than $\frac{n+t}{2}$ simulators. This happens strictly before the first READ'ANSWER (z, y, \perp) message is sent by a correct simulator. Since $|Q_{i,vw,r}|, |Q_{j,vw,r}| > \frac{n+t}{2}$, $Q_{i,vw,r} \cap Q_{j,vw,r}$ contains at least a correct simulator p_k .

Simulator p_k thus broadcast a READ'ANSWER (y, z, \perp) message and a READ'ANSWER (z, y, \perp) message (line B26). It then had $views_k[z] = \perp$ before broadcasting the READ'ANSWER (y, z, \perp) message and $views_k[y] = \perp$ before broadcasting the READ'ANSWER (z, y, \perp) . Because of the first instruction of line B25 this is impossible, and thus each correct process that sends a VIEW'WITNESS $(z, all_views_i[z])$ message ended the wait instruction of lines B38-B39 by verifying the predicate of line B38. This entails that $\forall x \in \Pi : all_views_x[z] \neq \perp \Rightarrow all_views_x[z][y] \neq \perp$. Consequently, $all_views_i[z][y] \neq \perp$.

Since $z \in \min_ \sigma_i$, $all_views_i[z] \neq \perp$ and thus $all_views_i[z][y] \neq \perp$. According to the predicate of line C16, this entails that $y \in \min_ \sigma_i$, and since the previous reasoning holds for any $y \in \min_ \sigma_j$, it shows that $\min_ \sigma_j \subseteq \min_ \sigma_i$. It follows that, when q_i executes line C17, $\forall y \in \min_ \sigma_j : all_views_i[y] \neq \perp$ and, consequently, $\forall y \in \min_ \sigma_j : all_views_i[y] = all_views_j[y]$. It entails that if $|\min_ \sigma_j| < |\min_ \sigma_i|$, then $\min_ \sigma_j$ would have been chosen by q_i at line C17, which proves that $\min_ \sigma_i = \min_ \sigma_j$ and contradicts the fact that q_i and q_j decide differently. □ Lemma 18

Correct values are valid

Lemma 19. *If a correct simulator q_i decides the value v , there is a correct simulator q_j such that $valid_j(-, v)$.*

Proof

Let v be the value decided by a correct simulator q_i . Value v has then be proposed by a simulator q_j such that $all_views_i[j] \neq \perp$ (definition of σ at lines 15-C16 and choice of value at line C18). In order to assign a non- \perp value to $all_views_i[j]$, q_i must have received VIEW'WITNESS $(j, -)$ messages from more than $\frac{n+t}{2}$ different simulators (lines B47-B48), and consequently from at least one correct simulator. Consider the first correct simulator q_x that has broadcast a VIEW'WITNESS $(j, -)$ message. Before sending it, it must have assigned a non- \perp value to $values_x[j]$ (lines B35-B42). It then has received either (a) VALUE'WITNESS $(j, -)$ messages from more than $\frac{n+t}{2}$ different simulators or (b) VALUE'ACK $(j, -)$ messages from more than $\frac{n+t}{2}$ different simulators.

In case (a), consider the first correct simulator q_k that has broadcast a VALUE'WITNESS $(j, -)$ message. In order to do so, it must have received VALUE'VALID $(j, -)$ messages from more than $\frac{n+t}{2}$ different simulators (lines B16-17). The predicate $valid_k(j, v)$ must have been satisfied at the simulators that broadcast these messages (line B15). In case (b), the first correct simulator that has broadcast a VALUE'ACK $(j, -)$ message must first have received VALUE'WITNESS $(j, -)$ messages from more than $\frac{n+t}{2}$ different simulators (lines B21-B23). The situation is then similar to Case (a). □ Lemma 19

Theorem 4. *The algorithms described in Figure 3 and Figure 4 implement a safe-agreement object in $BAMP_{n,t}[t < n/3]$.*

Proof The proof follows from the previous lemmas. □ Theorem 4

5.6 Simulation algorithm and its proof in $BAMP_{n,t}[t < n/3]$

Simulation algorithm When we consider the simulation algorithm described in Figure 2, we observe that the n simulators communicate only through safe agreement objects. It follows that the same algorithm works

in $\mathcal{BAMP}_{n,t}[t < n/3]$, when the crash-tolerant safe agreement objects are replaced by Byzantine-tolerant safe agreement objects previously described. Two things remain to be done: define the specific sub-predicate $P2$ of the predicate $\text{valid}()$, and do a specific proof of this algorithm (i.e., a proof based on the specification of the Byzantine-tolerant safe agreement objects defined in Section 5.2).

Sub-predicate $P2$ As far as $P2$ is concerned we have the following. Let us consider the simulator q_i that invokes $\text{valid}_i(j, v)$, with respect to the simulation of a process p_x . In the simulation algorithm, the parameter v is the message msg that q_j proposes to a safe agreement object from which will be decided the next message to be received by the simulated process p_x (lines 08-09 of Figure 2). $P2$ checks, from q_i 's local point of view, that, if the message v has been sent in the simulation, then it has not yet been consumed, i.e., $(v \in \text{sent}_i[x]) \Rightarrow (v \notin \text{received}_i[x])$.

Proof of the simulation algorithm in $\mathcal{BAMP}_{n,t}[t < n/3]$

Lemma 20. *The simulation of at most t simulated processes can be blocked.*

Proof The only places where a correct simulator q_i can block is during the invocation of the safe agreement operation $\text{decide}()$. Such invocations appear at line 02, and line 11.

Because the invocations by all the correct simulators of $\text{decide}()$ on all the safe agreement objects terminate, except for at most t safe agreement objects (Lemma 14), the simulation of at most t simulated processes can be blocked. \square Lemma 20

Lemma 21. *The simulation of the reception of the k -th message received by a simulated process p_j , returns the same message at all correct simulators.*

Proof The simulation of the message receptions for a simulated process p_j , are executed at each correct simulator q_i at lines 08-11, and all the correct simulators use the same sequence of sequence numbers (line 07). It then follows from the agreement property of the safe agreement object $SA[j, sn]$, that no two correct simulators obtain different messages when they invoke $SA[j, sn].\text{decide}()$, and the lemma follows. \square Lemma 21

Lemma 22. *For every simulated processes p_j , no two correct simulators return different values.*

Proof The only non-deterministic elements of the simulation are the input vectors $\text{input}_i[1..n']$ at each simulator q_i , and the reception of the simulated messages.

The lines 01-02 of the simulation force the correct simulators to agree on the same input value for each simulated process p_j , $1 \leq j \leq n'$. Similarly, as shown by Lemma 21, for each simulated process p_j , the lines 07-11 direct the simulators to agree on the very same sequence of messages received by p_j . It follows from the fact that the function $\delta_j()$ is deterministic, that any two correct simulators q_i and q_k , that execute lines 15-16 during the same “round number” $sn_i[j] = sn_k[j]$, are such that $\text{state}_i[j] = \text{state}_k[j]$, from which the lemma follows. \square Lemma 22

Lemma 23. *The sequences of message receptions simulated by each simulator q_i on behalf of each simulated process p_j , define a correct execution of the simulated algorithm.*

Proof To prove the correctness of the simulation, we have to show that

1. Every message that was received by a simulated process was sent by another simulated process,
2. Every message that was sent by a simulated process to another simulated process (whose simulation is not blocked either), is received, and
3. The simulated messages respect a simulated physical order (i.e., no message is “received” before being “sent”).

Item 1 follows from Lemma 19 and from the definition of $P2$. Item 2 is satisfied because the messages sent by the simulated process p_j to the simulated process p_k are received (lines 09-11) in their sending order (as defined at line 04 and line 14). Hence, if p_k is not blocked (due to a faulty simulator) it obtains the messages from p_j in their sending order.

For Item 3, let us define a (simulated) physical order as follows. For each simulated message m , let us consider the first time at which the reception of m was simulated (i.e., this occurs when –for the first time– a simulator terminates the invocation of $SA[-, -].\text{decide}()$ that returns m). A message that is decided has been proposed by a simulator to a safe agreement object before being decided (validity property). The sending time of a simulated message is then the first time at which $SA[-, -].\text{propose}(m)$ is invoked by a simulator. It follows that any simulated message is sent before being received, which concludes the lemma. \square *Lemma 23*

Lemma 24. *Each correct simulator q_i computes the decision value of at least $(n' - t)$ simulated processes.*

Proof Due to Lemma 20, and the fact that at most t simulators may be byzantine, it follows that at most t simulated processes may be prevented from progressing. As (a) by assumption the simulated algorithm A' is t -resilient, and (b) due to Lemma 23 the simulation produces a correct simulation of A' , it follows that at least $(n' - t)$ simulated processes decide a value. \square *Lemma 24*

Theorem 5. *Let A be an algorithm solving a decision task in $\text{CAMP}_{n',t}[t < n']$. The algorithm described in Figure 2, in which Byzantine-tolerant safe agreement objects are used, is a correct simulation of A in $\text{BAMP}_{n,t}[t < n/3]$.*

Proof The theorem follows from Lemma 23 and Lemma 24. \square *Theorem 5*

Additionally, the reader can easily check that the simulation of a message only requires a polynomial number of messages in the base system, and the increase in size of these messages, when compared to the size of the simulated message, is also polynomial.

6 Implications of the Simulation

BG-simulation in Byzantine message-passing systems A main result of this paper is a signature-free distributed algorithm that solves BG-simulation in Byzantine asynchronous message-passing systems. In addition to being the first algorithm that solves BG-simulation in such a severe failure context, the proposed simulation algorithm has noteworthy applications as shown below.

From Byzantine-failures to crash failures in message-passing systems The simulation presented here allows the execution of a t -resilient crash-tolerant algorithm in an asynchronous message-passing system where up to t processes may be Byzantine. A feature that is sometimes required from a Byzantine-tolerant algorithm solving a task (not usually considered in the crash failure case) is that the value decided by any correct process should be based only on inputs of correct processes. This prevents Byzantine processes from “polluting” the computation with their inputs. A way to guarantee that an input has been proposed by a correct process is to check that it has been proposed by at least $(t + 1)$ different processes. Assuming that in any execution at most m values are proposed, this constraint translates as $n - t > mt$ [16, 28].

In the case of the simulation presented in Section 5, this requirement can easily be satisfied by adding a first step of computation before the start of the simulation. Simulators first broadcast their input. They then echo every value that they receive from more than $t + 1$ different simulators, and consider these values (and only these values) as valid inputs. An input considered valid by a correct simulator is then eventually considered valid by all correct simulators, and the only inputs allowed in the simulation are inputs of correct simulators. Because we consider colorless tasks, the choice of output is done in the same way as in the original BG-simulation: a simulator can adopt the output of any simulated process that has decided a value.

The possible Byzantine behaviors are restrained by the underlying Byzantine-tolerant safe agreement objects used in the simulation. Surprisingly, this shows that, from the point of view of the computability of colorless tasks and assuming $n > (m + 1)t$ (this requirement always implies $n > 3t$ when at least two different values can be proposed), Byzantine failures are equivalent to crash-failures. This provides us with a new understanding of Byzantine failures and shows that their impact can be restricted to the much simpler crash-failure case.

From wait-free shared memory to message-passing The proposed simulation can be combined with previous works to further extend the scope of the result. Consider an algorithm A_0 that solves a colorless task, where $m > 1$, in a wait-free read/write memory system of $t + 1$ processes, denoted $\mathcal{CARW}_{t+1,t}[\emptyset]$. Using the basic BG-simulation [6], this algorithm can be transformed into an algorithm A_1 that works in the t -resilient read/write memory system of $(m + 1)t + 1$ processes, in which at most t can crash. This model is denoted $\mathcal{CARW}_{(m+1)t+1,t}[\emptyset]$. Using an implementation of a read/write memory in a crash-prone message-passing system in which a majority of processes are correct [2], we obtain an algorithm A_2 which work in $\mathcal{CAMP}_{(m+1)t+1,t}[\emptyset]$ (message-passing system system of $(m + 1)t + 1$ processes, in which at most t can crash; notice that $m > 0 \Rightarrow (m + 1)t + 1 > 2t$). Finally, using the simulation presented in this paper, we obtain Byzantine-tolerant algorithm A_3 which works in $\mathcal{BAMP}_{(m+1)t+1,t}[\emptyset]$ (message-passing system of $(m + 1)t + 1$ processes, of which at most t can be Byzantine; notice that $m > 1 \Rightarrow (m + 1)t + 1 > 3t$).

These transformations show that, as far as the computability of colorless tasks that admit up to $m > 1$ different input values is concerned, an n -process Byzantine-prone message-passing system, in which up to $t < n/(m + 1)$ processes can be Byzantine, is equivalent to a wait-free shared memory system of $t + 1$ processes, which at most commit crash failures. When considering colorless tasks with $m > 1$, a figure relating these transformations is depicted in Figure 6. Differently from the full-information algorithm presented in [28], the simulation presented in the present paper (along with [6] and [2]) allows a *direct* transformation of any wait-free shared-memory algorithm that solves a colorless task into a message-passing Byzantine-tolerant algorithm.

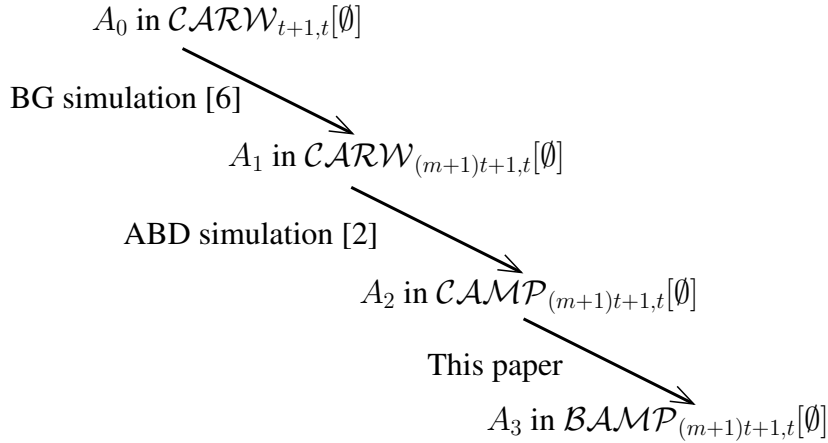


Figure 6: From crash in read/write to Byzantine in message-passing (with $m > 1$)

Acknowledgments

This work has been partially supported by the Franco-German DFG-ANR Project 40300781. This project (named DISCMAT) is devoted to mathematical methods in distributed computing. The authors would like to thank Sergio Rajsbaum for discussions on the BG simulation.

References

- [1] Afek Y., Attiya H., Dolev D., Gafni E., Merritt M. and Shavit N., Atomic snapshots of shared memory. *Journal of the ACM*, 40(4):873-890, 1993.
- [2] Attiya H., Bar-Noy A. and Dolev D., Sharing memory robustly in message passing systems. *Journal of the ACM*, 42(1):121-132, 1995.
- [3] Attiya H., Bar-Noy A., Dolev D., Peleg D. and Reischuk R., Renaming in an asynchronous environment. *Journal of the ACM*, 37(3):524-548, 1990.
- [4] Attiya H. and Welch J., *Distributed computing, fundamentals, simulation and advanced topics (Second edition)*. Wiley Series on Parallel and Distributed Computing, 414 pages, 2004.
- [5] Bazzi R. A., and Neiger G., Optimally simulating crash failures in a Byzantine environment. *Proc. 5th International Workshop on Distributed Algorithms (WDAG'91)*, Springer LNCS 579, pp. 108-128, 1991.
- [6] Borowsky E. and Gafni E., Generalized FLP impossibility results for t -resilient asynchronous computations. *Proc. 25th ACM Symposium on Theory of Computing (STOC'93)*, ACM Press, pp. 91-100, 1993.
- [7] Borowsky E., Gafni E., Lynch N. and Rajsbaum S., The BG distributed simulation algorithm. *Distributed Computing*, 14:127-146, 2001.
- [8] Bracha G., Asynchronous Byzantine agreement protocols. *Information & Computation*, 75(2):130-143, 1987.
- [9] Castañeda A., Rajsbaum S., New combinatorial topology bounds for renaming: the upper bound. *Journal of the ACM*, 59(1), article 3, 49 pages, 2012.
- [10] Castañeda A., Rajsbaum S., and Raynal M., The renaming problem in shared memory systems: an introduction. *Computer Science Review*, 229-251, 2011.
- [11] Chaudhuri S., More *choices* allow more *faults*: set consensus problems in totally asynchronous systems. *Information and Computation*, 105:132-158, 1993.
- [12] Coan B.A., A compiler that increases the fault-tolerance of asynchronous protocols. *IEEE Transactions on Computers*, 37(12):1541—1553 (1988).
- [13] Fischer M.J., Lynch N.A. and Paterson M.S., Impossibility of distributed consensus with one faulty process. *Journal of the ACM*, 32(2):374-382, 1985.
- [14] Gafni E., The extended BG simulation and the characterization of t -resiliency. *Proc. 41th ACM Symposium on Theory of Computing (STOC'09)*, ACM Press, pp. 85-92, 2009.
- [15] Goldreich O., Micali S. and Wigderson A., How to play any mental game or a completeness theorem for protocols with honest majority. *Proc. 19th ACM Symposium on Theory of Computing (STOC'87)*, ACM Press, pp. 218-229, 1987.
- [16] Herlihy M.P., Kozlov D., and Rajsbaum S., *Distributed computing through combinatorial topology*, Morgan Kaufmann/Elsevier, 336 pages, 2014 (ISBN 9780124045781).
- [17] Herlihy M. and Rajsbaum S., The decidability of distributed decision tasks (Extended Abstract). *Proc. 29th ACM Symposium on Theory of Computing (STOC'97)*, ACM Press, pp. 589-598, 1997.
- [18] Herlihy M., Rajsbaum S., and Raynal M., Power and limits of distributed computing shared memory models. *Theoretical Computer Science*, 509:3-24, 2013.
- [19] Herlihy M., Shavit N., The topological structure of asynchronous computability. *Journal of the ACM*, 46(6):858-923, 1999.
- [20] Ho Ch., Dolev D. and van Renesse R., Making distributed applications robust. *Proc. 11th Int'l Conference on Principles of Distributed Computing (OPODIS'15)*, Springer LNCS 4878, pp. 232-246, (2007).
- [21] Imbs D., Rajsbaum S., and Raynal M., The universe of symmetry breaking tasks. *Proc. 18th Int'l Colloquium on Structural Information and Communication Complexity (SIROCCO'11)*, Springer, LNCS 6796, pp. 66-77, 2011.

- [22] Imbs D. and Raynal M., Visiting Gafni's Reduction Land: From the BG simulation to the extended BG simulation. *Proc. 11th Int'l Symposium on Stabilization, Safety, and Security of Distributed Systems (SSS'09)*, Springer LNCS 5873, pp. 369-383, 2009.
- [23] Imbs D. and Raynal M., The multiplicative power of consensus numbers. *Proc. 29th ACM Symposium on Principles of Distributed Computing (PODC'11)*, ACM Press, pp. 26-35, 2011.
- [24] Imbs D. and Raynal M., A Liveness Condition for Concurrent Objects: x -Wait-Freedom. *Concurrency and Computation: Practice and Experience*, 23(17): 2154-2166, 2011.
- [25] Lamport L., On Interprocess communication, Part I: Basic formalism. *Distributed Computing*, 1(2):77-85, 1986.
- [26] Lamport L., Shostack R., and Pease M., The Byzantine generals problem. *ACM Transactions on Programming Languages and Systems*, 4(3):382-401, 1982.
- [27] Loui M.C., and Abu-Amara H.H., Memory requirements for agreement among unreliable asynchronous processes. *Par. and Distributed Computing: vol. 4 of Advances in Comp. Research*, JAI Press, 4:163-183, 1987.
- [28] Mendes H., Tasson Ch., and Herlihy M., Distributed computability in Byzantine asynchronous systems. *Proc. 46th Symposium on the Theory of Computing (STOC'14)*, ACM Press, pp. 704-713, 2014.
- [29] Neiger G., and Toueg S., Automatically increasing the fault-tolerance of distributed algorithms. *Journal of Algorithms*, 11(3): 374-419, 1990.
- [30] Saks M. and Zaharoglou F., Wait-free k -set agreement is impossible: the topology of public knowledge. *SIAM Journal on Computing*, 29(5):1449-1483, 2000.
- [31] Srikanth T.K. and Toueg S., Simulating authenticated broadcasts to derive simple fault-tolerant algorithms. *Distributed Computing*, 2(2):80-94 (1987).